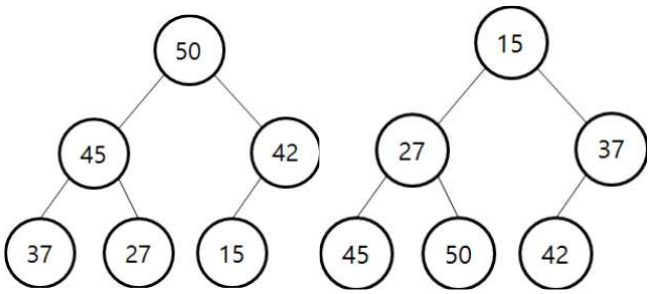
<알고리즘 Mix Tape 04>

3장_추가 힙(Heap) 정렬(170612)

실제로 최댓값과 최솟값을 찾는데 있어서 선형적으로 탐색을 하면 O(n) 만큼 오래 걸리게 된다. 하지만 힙 정렬을 이용해서 최댓값과 최솟값을 힙 자료구조를 이용하여 O(log n)의 시간 복잡도로 정렬을 하는 방법이 바로 힙 정렬 알고리즘이다. 이에 대해서 간단하게 짚고 넘어가보도록 하자.

1. 힙(Heap) 정렬



힙(Heap)은 이진트리의 일종으로 볼 수 있다. 이진 탐색 트리에서는 작으면 왼쪽, 크면 오른쪽으로 밀어 붙이는 특성을 이용했지만 힙은 왼쪽, 오른쪽이 중요한 것이 아니라 레벨(즉 Heap 노드의 깊이가 되겠죠??)의 영향이 꽤 크다. 이 두 가지의 트리가 각각 최대 힙, 최소 힙이다. 힙 자료구조에 대하여 잠깐 알아보도록 하고, 그 다음에 힙 정렬이 어떤 원리로 돌아가는지에 대해 공부를 해보자.

1-1) Heap 자료구조

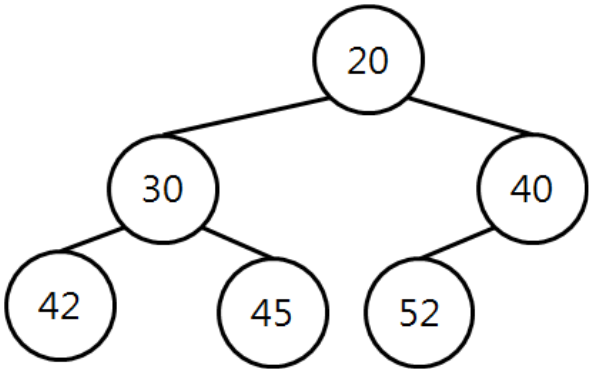
Heap을 만족하기 위해서는 아래와 같은 조건들을 성립해야 한다.

-> 각 레벨(높이) 별로 대소 관계가 뚜렷해야 한다. 이것이 Heap의 궁극적인 목표이다.

-> 힙은 꼭 완전 이진트리여야 한다. 즉 자식들이 왼쪽부터 정리가 되어 있어야 한다는 뜻이다. 또한 포화 이진트리(쉽게 이야기하면 알배기 트리?? 한 레벨에 자식들이 가득 차 있는 트리)도 완전 이진트리의 일부니깐 포화 이진트리여도 상관은 없다.

-> 노드의 값은 자식 노드들의 값의 대소 관계에 대해 철저히 지켜야 한다. 최대 힙인 경우에는 부모 노드가 크게끔 만들고, 최소 힙은 부모 노드가 작게끔 만들라는 뜻이다.

이제 Heap 자료구조에 대해서 잠깐 살펴보도록 하자. Heap은 아시다시피 완전 이진트리로 표현을 해야 하기 때문에 Node를 이용해서 개고생하지 말고 여기서만 잠깐 배열을 이용하도록 하자.



이러한 Heap 트리는 아래와 같이 정리할 수 있겠다.

a[]	20	30	40	42	45	52	

Heap은 배열을 통해서 정리를 하는 원리로 작동하기 때문에 인덱스가 엄청 중요하다는 점을 느낄 수 있을 것이다. 실제로 인덱스는 아래와 같은 방법으로 부여가 된다.

-> root 노드의 인덱스는 0이다.

-> n번째 인덱스의 자식 노드들은 2n+1, 2n+2이다.

-> n번째 인덱스의 부모 노드는 [(n-1)/2]이다.(단 n은 0이면 안 된다. 즉 root 노드가 아니면 된다.) 여기서 [x]는 가우스 함수로 소수점을 제외한 정수를 가져오라는 의미로 생각하면 되겠다.

이러한 조건으로 만들어진 힙 자료구조에 대해 슬슬 알았으니 힙 정렬 알고리즘에 대해 공부해보도록 하자.

1-2) Heap 정렬 알고리즘

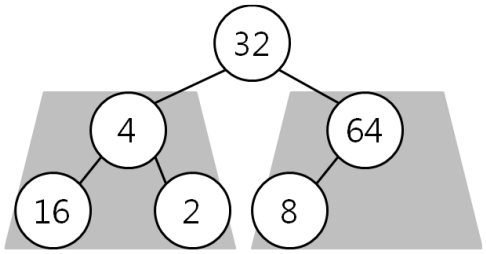
```
import java.util.Arrays;
public class Example1 {
  static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
  static void buildHeap(int[] a) {
    int end = a.length - 1;
    for (int i = end / 2; i >= 0; --i)
      heapify(a, i, end);
  }
  static void heapify(int[] a, int k, int
end) {
    int left = 2 * k + 1, right = 2 * k + 2;
    int smaller;
    if (right <= end)
      smaller = (a[left] < a[right]) ? left :
right;
    else if (left <= end) smaller = left;
    else return;
    if (a[smaller] < a[k]) {
      swap(a, smaller, k);
      heapify(a, smaller, end);
    }
  }
  static void heapSort(int[] a) {
    buildHeap(a);
    for (int end = a.length - 1; end >= 1;
--end) {
      swap(a, 0, end);
      heapify(a, 0, end - 1);
    }
  }
  public static void main(String[] args) {
    int[] a = { 32, 4, 64, 16, 2, 8 };
    heapSort(a);
    System.out.println(Arrays.toString(a));
  }
}
```

이 코드를 잠깐 살펴보는 기회를 가져보도록 하자. 이번에는 2의 제곱들로 이루어진 숫자들을 추가함으로서 Heap이 어떤 과정으로 이루어지는지에 대하여 알아보도록 하자.

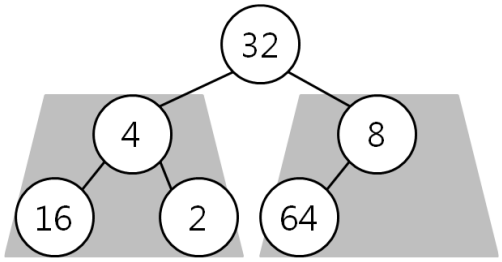
-> Next Page ->

A단계) 힙을 형성하자

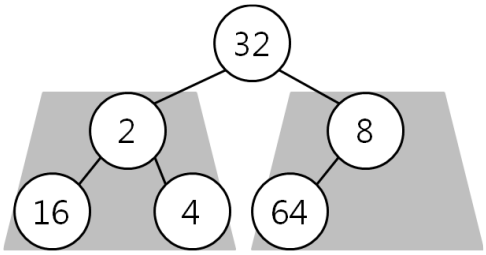
일단 강의 노트에 나온데로 힙을 구성하는 것은 Heap 자료구조를 이루는 이야기 밖에 안 된다. 실제로 heapify 메소드 내부에서 어떻게 형성이 되는지에 대해서 이해를 할 필요가 있다.



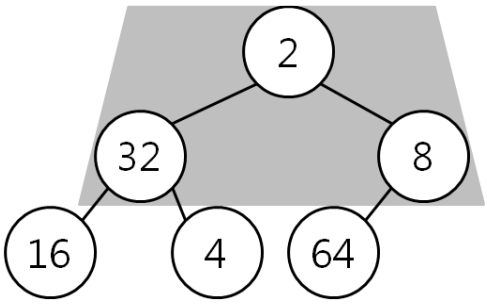
-> 일단 32의 우측 서브 트리인 64부터가 Heap를 만족하는지에 대해 따져봐야 된다. 그렇지만 8이 아래에 있으니 만족한다고 볼 수 없겠다. 아래처럼 조정을 할 요가 있다.



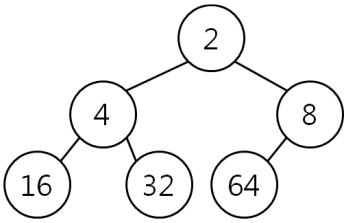
-> 이제 32의 좌측 서브 트리인 4부터 Heap를 만족하는지에 대해 따져보도록 한다. 일단 서브 트리의 자식들의 값들 중에서 가장 작은 노드는 2가 되는데 서브 트리의 루트 노드와 비교를 하면 작기 때문에 조정을 할 필요가 있다. 그래서 2랑 4끼리 자리를 바꿔주도록 하자.



-> 그리고 최종적으로 루트 노드(32)의 자식들을 비교를 하면 2랑 8중에서 분명히 작은 값은 2이다. 그리고 루트 노드랑 자식 노드(2) 중에서 작은 값은 2이니 2를 루트 노드로 보내주도록 한다.



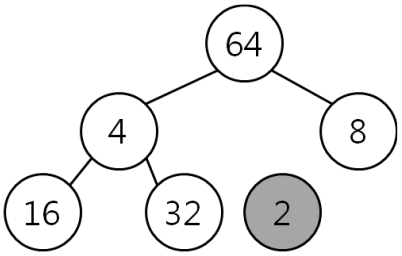
-> 루트 노드만 다 정렬한다고 Heap 정렬이 완료된 것이 절대로 아니다. 왜냐면 32는 분명 4보다 크고, 논리적으로 따지고 보면 힙이 아니기 때문이다. 최종적으로 32란 자식 노드와 4랑 비교를 해서 Heap을 완성을 해서 정렬을 하는 원리로 생각을 하면 되겠다.



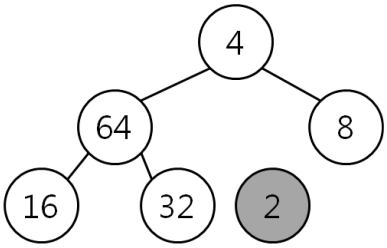
이제 힙이 형성되었으니 실제로 힙은 어떻게 정렬이 되는지에 대해 알아볼 필요가 있다.

B단계) 최솟값을 빼내면서 힙 정렬을 실행하자

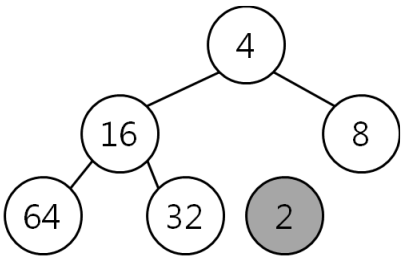
B-1) 2를 빼내도록 하자



2랑 마지막 인덱스에 존재하는 64랑 맞바꿈을 한다. 실상 2는 값이 나왔으니 Heap에서 존재감이 없어진 거랑 다를 바 없다. 그럼 64가 루트로 올라왔으니 자식인 4랑 8를 비교하면 4가 제일 작으니 4를 위로 올려보내면 되겠다.

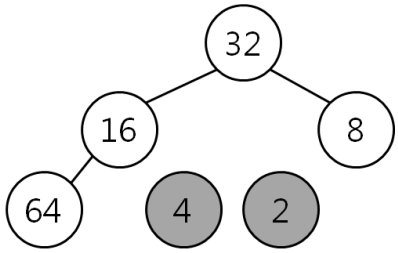


그럼 이제 64는 내려왔으니 16이랑 32랑 비교를 해서 가장 작은 값인 16이랑 맞바꿈을 해주도록 한다.

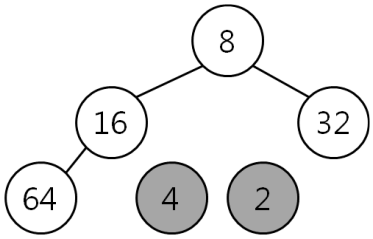


그럼 Heap 자료구조를 만족하는 범위 이내에서 2라는 값이 정리가 완료되었다.

B-2) 4를 빼내도록 하자

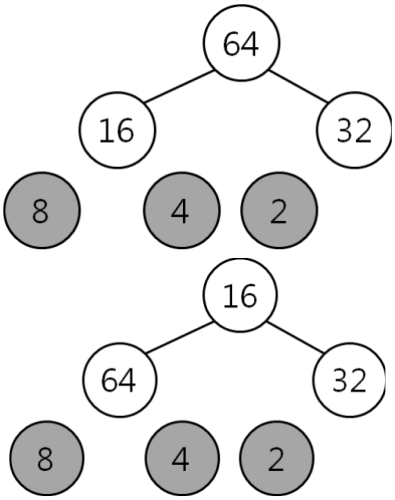


일단 4를 빼내게 된다면 분명 마지막 인덱스에 존재하는 32랑 맞바꿈이 될 것이다. 그래서 32에 대해서 비교를 하면서 힙을 완성을 하면 되는데 32의 자식들인 8이랑 16을 비교를 해서 가장 작은 8이랑 바꾸면 되겠다.



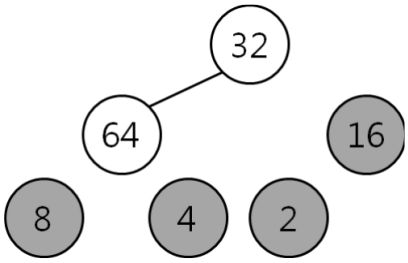
그럼 Heap을 지켜내는 범위 이내에서 종결이 된다.

B-3) 8을 빼내도록 하자



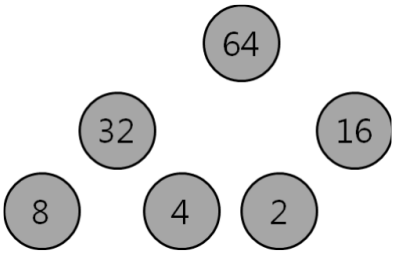
8이랑 마지막 인덱스에 존재하는 64랑 맞바꿈을 해준다. 그러면 64가 맨 위로 올라오게 되면서 64의 자식들인 16이랑 32를 비교해서 Heap을 만족하게 한다.

B-4) 16을 빼내도록 하자



마지막 인덱스에 있는 32랑 맨 위에 있는 16이랑 인덱스를 바꾸면? 어라? 그냥 Heap을 만족한다.

B-5) 32를 빼고 64를 빼면 Heap 정렬이 끝난다.

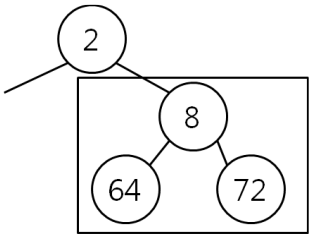


이제 32랑 64를 서로 바꾸게 되면 최댓값들이 올라오게 되면서 Heap 정렬의 과정이 끝났다. 그럼 현재 배열들의 목록들을 빼보면 64, 32, 16, 8, 4, 2가 나오게 되어 정상 종료가 되는데 시간 복잡도는 O(n log n)이 나온다.
-> heapSort 내 반복문의 시간 복잡도 : O(n)
-> heapify의 시간 복잡도 : O(log n)
이 두 복잡도를 콜라보해보면 이해할 것이다.

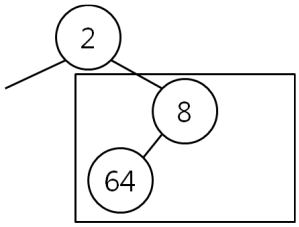
1-3) heapify 메소드 분석

```
static void heapify(int[] a, int k, int end){
  int left = 2 * k + 1, right = 2 * k + 2;
 // 1
  int smaller;
  if (right <= end)
    smaller = (a[left] < a[right]) ? left : right;
// 2
  else if (left <= end) smaller = left; // 3
  else return; // 4
  if (a[smaller] < a[k]) {
    swap(a, smaller, k);
    heapify(a, smaller, end);
  } // 5
}
```

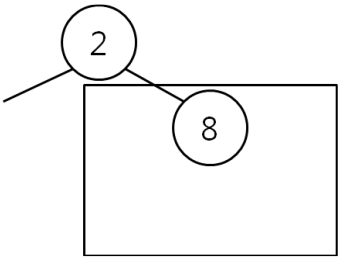
1 : 일단 left랑 right에 대한 의미를 알고 넘어가야 한다. heap에서 자식들에 대한 인덱스를 알고 싶다면 현재 인덱스가 되는 k에 대해서 2k+1은 좌측 아들, 2k+2는 우측 아들이 되겠다. 그런데 실제로 end랑 비교를 해야 하는데 이는 2, 3, 4를 통해 분기가 된다.



2 : 이 문장은 자식들의 수가 영향을 주는 요소가 있는데 아까 같이 2부터 64까지 저장된 요소에 72를 추가하게 되면 7개가 된다. 즉 완전 이진트리를 만족하되 인덱스로 표현하면 마지막 부모의 자식이 2개인가를 만족하는가에 대해 따진다고 생각하면 된다. 그러면 왼쪽과 오른쪽과 비교를 해서 작은 값이 결국 smaller 인덱스가 된다.



3 : 이는 완전 이진트리를 만족하는 범위에서 마지막 부모의 자식이 1개인 점에 대해 따진다고 생각하면 된다. 그러면 결국 smaller는 왼쪽 자식 밖에 존재하지 않는다.



4 : 완전 이진트리를 만족하는 범위에서 마지막 부모의 자식을 비교를 하되, 둘 다 없다면 재귀 호출을 벗어나는 의미로 생각을 하면 되겠다. 4번 문장에 대해서는 Heap 정렬을 하게 된다면 대개 root 노드에 대한 정렬부분에서만 쓰이고 많이 안 쓰게 된다.

5 : 최종적으로 smaller 인덱스랑 현재 인덱스 k랑 비교를 해서 heap 구조를 만족을 할 수 있도록 재귀 문장을 통해 호출을 시킨다.